DOI:
10.1287/isre.2016.0646

Document Version
Peer reviewed version

FOLDING AND UNFOLDING:
BALANCING OPENNESS AND TRANSPARENCY IN OPEN SOURCE COMMUNITIES

Maha Shaikh
Warwick Business School
University of Warwick
Coventry
United Kingdom
E-mail: maha.shaikh@wbs.ac.uk

Emmanuelle Vaast
Desautels Faculty of Management
McGill University
Montreal, Quebec
Canada
E-mail: emmanuelle.vaast@mcgill.ca

Abstract

Open source communities rely on the espoused premise of complete openness and transparency of source code and development process. Yet, openness and transparency at times need to be balanced out with moments of less open and transparent work. Through our detailed study of Linux Kernel development we build a theory that explains that transparency and openness are nuanced and changing qualities that certain developers manage as they use multiple digital technologies and create, in moments of needs, more opaque and closed digital spaces of work. We refer to these spaces as digital folds. Our paper contributes to extant literature: by providing a process theory of how transparency and openness are balanced with opacity and closure in open source communities according to the needs of the development work; by conceptualizing the nature of digital folds and their role in providing quiet spaces of work: and, by articulating how the process of digital folding and unfolding is made far more possible by select elite actors' navigating the line between the pragmatics of coding and the accepted ideology of openness and transparency.

Keywords: Open source communities, digital folds, transparency, openness, opacity, closure, software development work, qualitative research, archival data, ideology.

INTRODUCTION

Growing scholarly and topical interest in online communities and, in particular, in open source communities has given rise to a wealth of research. Aspects such as how open source communities emerge (Endres et al. 2007; Oh and Jeon 2007; von Krogh et al. 2003), how they are governed (O'Mahony and Ferraro 2007; Sharma et al. 2002; Tullio and Staples 2014), and how they coordinate work and practices (Ben-Menahem et al. 2015; Crowston and Howison 2006; Howison and Crowston 2014; Koch and Schneider 2002; von Hippel and von Krogh 2003) have been fruitfully explored. Other facets of open source that have seen deep interest include the licensing schemes and their impact on the code and community (Laurent 2004; Lerner and Tirole 2005; Olson 2005; Singh and Phelps 2013), the economics of how open source functions and how it is even made possible (Benkler 2002; Benkler 2004; Bonaccorsi and Rossi 2003; Lerner and Tirole 2002), and development practices (Crowston and Howison 2006; Feller and Fitzgerald 2002; Feller et al. 2002; Fitzgerald and Feller 2002; Raymond 1999).

Over the years, there has been a gradual but clear shift in open source communities from what was originally seen as a group of self-selecting volunteers to situations where many within a community are employed by commercial organizations to do the company's work within a community. This has brought renewed academic interest to issues such as developers' motivations (Oram 2011; Spaeth et al. 2014; von Krogh et al. 2012), a topic that had seen much early interest as well (Hars and Ou 2002; Hertel et al. 2003; Lakhani and Wolf 2005; Shah 2006). More generally, these new dynamics of open source communities have led researchers to question some taken for granted ideas about how open source communities work and organize their activities. It is one such topic - that of the balancing of openness and transparency with moments of closure and opacity in open source development work - that we draw attention to and aim to contribute to in this paper.

The principles of transparency and openness are at the core of the rhetorical discourse of open source communities (Hertel et al. 2003; Lakhani and Wolf 2005). Openness and transparency, though highly related concepts, are distinct. Openness is primarily concerned with the issue of accessibility, where access to the code (product) and the various parts of the process of development are available to all (Dahlander and Gann

2010; MacCormack et al. 2006; von Krogh and von Hippel 2006; West 2003). Transparency relates to the visibility of code, process, techniques, and communications (Coleman 2004; Gacek and Arief 2004; Jorgensen 2001; MacCormack et al. 2006; Stewart et al. 2006). Espousing the principles of transparency and openness constitutes a key "*deep defining structure,*" or ideology, that shapes much of the framing discourses in and about open source communities (Barrett et al. 2013). Yet, transparency and openness have been sparsely examined or questioned in open source development research. Therefore we still know precious little about the process through which developers actually navigate openness and transparency in open source development work.

We focus on a high profile open source case, the Linux Kernel development. We followed nearly a decade's worth of online interactions in this community. Our attention was initially drawn to a particular discussion on version control software (VCS) that held serious developer attention. We examined discussions in the Linux Kernel Mailing List (LKML) and discovered that developers relied upon the use of overlapping technologies in ways that created small and temporary pockets of less open and transparent development work. In other words, as developers at times used technologies they also created "*the knot, the fold, where order and disorder meet*" (Clegg et al. 2005, p154). Some developers met in these spaces to address issues that had so far not been solved by the community in the open. These spaces that seemed to contradict the principles of complete openness and transparency in open source development work could be generative for the development process. Working temporarily in these secluded digital spaces helped some members of the community come up with solutions to technical issues when transparency and openness had fallen short.

This paper examines how some developers may rely upon digital technologies that 'fold' over each other and create private pockets of interactions that may temporarily limit the transparency and openness of development work. Our study considers openness and transparency as fluid and manageable qualities in the open source development process. It examines the following question: *How do certain members of an open source community navigate the openness of the development process?*

Answering this question, this paper develops a theory of how select developers balance the principles of transparency and openness in open source with practical needs for opacity and closure by harnessing digital

technologies in ways that generate temporary protected spaces of work, i.e. digital folds. This work contributes to research in open source software communities a process theory of folding and unfolding. It adds to research on secluded spaces of work a conceptualization of digital folds as digitally-enabled, exclusive and provisional, spaces to accomplish a difficult, focused task. It also adds a conceptualization of how the espoused principles of openness and transparency can be mitigated, pragmatically, by occasional demands of work, and, ideologically, by the meritocratic principle.

The next section builds upon existing insights on openness and transparency in open source development. We then detail our methods, clarifying our data collection and analysis process. Our empirical findings focus on key processes of folding and unfolding in the Linux community. We build on these findings to establish our theory development, and then present implications of our work before concluding with a summary of this research, acknowledgement of its limitations, and avenues for exciting future studies.

TRANSPARENCY AND OPENNESS IN OPEN SOURCE COMMUNITIES

Open source communities seem to build on the very premise of the openness and transparency of development work (von Krogh and Spaeth 2007). These principles are usually taken for granted even by hackers (Raymond 1999) and are part of the deep structure of meanings that manifests in the discourse about open source (Dabbish et al. 2012; Ljungberg 2000; O'Reilly 1999; Raymond and Trader 1999). Some activists such as Richard Stallman have in this regard noted the criticality and vulnerability of these qualities in relation to software and have spoken of 'freedoms' that must be a part of all software used (Stallman 1984; 1999a; 1999b; 2002). These freedoms have been encapsulated and thus protected by the GNU General Public License (GPL) and the GNU Manifesto. The GPL tends to focus on freedoms made possible by access to the source of the code but there is less emphasis on an equally open development process. This can partly be explained by the fact that it is more difficult to articulate exactly what is meant by an open source process, and how open is open enough (West 2003).

Conversely, open source has given rise to related phenomena such as open data (Gurstein 2011; Streeter et al. 1996), open innovation (von Hippel and von Krogh 2003; West and Bogers 2014; West and Gallagher 2006), open government (Daffara and Gonzalez-Barahona 2010; Janssen et al. 2012; O'Reilly 2010), open

standards (West 2007), open hardware (Powell 2012), and open platforms (Boudreau 2010; Economides and Katsamakas 2006; Krishnamurthy and Tripathi 2009; West 2003). These phenomena are also built around openness and transparency as the dominant principles. In these cases researchers have considered the transparency and openness of the process more than the artefact or service under joint development. Yet, it does take openness of the process *and* of an open product/service to be truly open (Felin and Zenger 2014). It has however been noted that open source is a moving and changing phenomenon (Barrett et al. 2013; Deodhar et al. 2012; Spaeth et al. 2015; von Krogh et al. 2012) that has in part adapted to different commercial interests (Fitzgerald 2006). The shift in open source has seen an equivalent transformation in the underlying ideology that then affects the business models of adoption (Dahlander 2007; Deodhar et al. 2012), a profusion in licenses (Osterloh and Rota 2007; Scacchi and Alspaugh 2012), a change in development process (Dahlander and O'Mahony 2011; Fitzgerald 2006), and a deep change in ideological preferences (Campbell-Kelly and Garcia-Swartz 2009; Choi et al. 2015; Stewart and Gosain 2006). The latter work is attentive to ideology more particularly and provides a comprehensive characterization of the relationship between trust building in communities and ideology, where observance of beliefs, norms and values (elements of ideology) inspires trust and in turn a better retention rate of community members (Stewart and Gosain 2006). Trust is negotiated between members of the community where there is a difference in social status. How elite developers manage work and relationships with newbies affects the latter's desire to remain within this community (Stewart and Gosain 2006). Social status achievement has been explained as a process of moving from the periphery of a community to the central core over time (Dahlander and O'Mahony 2011), but also as an evaluation process where actors in the community weigh the worth of a fellow developer by looking to reputation cues that are publicly available (Stewart 2005). There is less literature to date that helps us to understand how ideas of meritocracy, elitism, and ideology are related in the everyday practices of development work.

A hybrid notion of open innovation, code and process (von Hippel and von Krogh 2003) has inspired much of the related literature. The concept of hybridization is interesting because it hints at the possibility of different forms co-existing. Therefore, actual development work in open source communities does not

always or necessarily rely upon entirely open and transparent processes. The private-collective innovation model provides a seminal argument explaining the nuanced mix of open and closed practices and possibilities for value creation and capture when there is a productive mix of open and closed and transparent and non-transparent (von Hippel and von Krogh 2003). Moreover, various layers of access, governance and possible exclusion may be exercised within a community (von Hippel and von Krogh 2003) just as in organizational hierarchies. Layers of management have been introduced into many open source projects, and the Linux Kernel case is no exception. This project in particular is known for its group of elite maintainers that are given the title of 'trusted lieutenants' (Dafermos 2001; Moody 2001; Schweik 2003; von Hippel and von Krogh 2003). The trusted lieutenant status, like any other designated role in open source communities, is earned by developers on the basis of contribution (usually code) and awarded on merit (Feller and Fitzgerald 2000). In order to have a patch of code accepted all submitting developers need to pass through certain steps of review, and one of the main ones is to persuade a trusted lieutenant that your code is valuable and does not break the system (Schweik 2003; von Hippel and von Krogh 2003; von Krogh et al. 2003). Meritocracy-based development communities, through such governance and coordination mechanisms reduce conflict and unnecessary politics (Kogut and Metiu 2001; Roberts et al. 2006).

Scaling down into more focused work on openness MacCormack et al (2006) make a strong case for an 'architecture for participation' in relation to code modularity. Participation, they argue, relies on access to software architecture, ability to reuse software, bug detection and fixing (Bagozzi and Dholakia 2006; MacCormack et al. 2006). This is reminiscent of a large body of work where openness and transparency of open source is unconditionally considered as positive. The use of tools like concurrent version systems (CVS), and publicly archived mailing lists offer the collective a trackable process for joint software development (Fogel 1999; Koch and Schneider 2000; Koch and Schneider 2002; Lakhani and von Hippel 2003; von Hippel and von Krogh 2003). Yet, organized work, whether it takes place in traditional or non-traditional settings, often involves processes that happen not only in public but also, at least at times, in more private settings. Such movement between public and private settings has implications for development work in open source communities but has not yet been much examined or theorized in the literature. The literature

to date has paid little theoretical attention to how such movement is made possible or to its significance for the development process. This is the under-investigated area that our research sets to explore and theorize. This area seems important to examine especially since, intuitively, one can imagine that extreme openness and transparency can be limiting and lead to interruptions and failures in decision making in the development process. It therefore stands to reason that at least some participants in open source development at times need to balance out the openness and transparency of the process with moments of closure and opacity for the development to work. Yet, how people navigate the openness of the open source development has not yet been investigated. What this balancing involves, who manages it, how it happens and with what consequences for the development process are important issues for open source development scholarship that deserve further investigation and theorizing.

Having presented the conceptual foundations that informed our thinking and helped us position our own work, we now turn to our methods and findings in order to further our conceptualization of how community members may navigate the openness and transparency of the context through what we call digital folds and the corresponding process of folding and unfolding.

METHODS

Research Setting

We opted for an in-depth, qualitative archival longitudinal study (Hoegl et al. 2004; Roberts et al. 2006; van Oorschot et al. 2013; Wright and Zammuto 2013). We chose the Linux Kernel community as our open source study because it has been in existence for over two decades, has shown sustainable growth in developer numbers, code base size, and users (commercial and otherwise) (Benkler 2002; Torvalds 1999), and has lived through a history of arguing over the various technologies needed to coordinate, organize and control the community (Lee and Cole 2003; Weber 2004). It may have been an extreme case (Flyvbjerg 2006; Yin 1981) with regard to its heated discussions over version control software use and choice but at the same time it offered a unique opportunity to unpack, through the words of the community, the ramifications of each technological choice and decision, and thus provided a strong theoretical sample (Eisenhardt

and Graebner 2007). Moreover, this case provided a high level of access to longitudinal data since the Linux community has carried on its discussions for over two decades.

Linux Kernel development is one of the largest software development projects where collaboration is essentially carried out online (Corbet et al. 2013). There are close to 10,000 developers working on this project which establishes it as one of the most active communities. Version control software is part of the basic eyeballing of bugs and patches that is encouraged in open source communities. It is part of the basic technical infrastructure that makes geographically dispersed development possible. Version control software (VCS) is a mechanism and software by which multiple versions of any software can be managed, kept track of and protected against overwriting (Clemm 1989; Kilpi 1997). Along with other digital technologies like email, and chat software (like IRC), version control software draws the community together and coordinates and organizes software development.

The Linux Kernel began in 1991 as a very small project initiated by Linus Torvalds, then a young Finnish student at the Helsinki University. Simple tools were used for at least the first 4-5 years of Linux Kernel development as the community was small enough to be managed by Torvalds on his own. He preferred the use of email for nearly all communications, coupled with IRC on occasion, for sending software patches to and fro between the community and himself. It was in mid-1995 that the community grew aware that a more sophisticated tool was becoming necessary to coordinate Linux work because patches were being overwritten or missed by Torvalds. A segment of the Linux community then began to use Concurrent Versions System (CVS) which was an open source version control system (Bar and Fogel 2003; Berliner 1990; Fogel 1999; Grune 2003) to manage Linux code and patch updates. Many were content with this digital tool but Torvalds found this tool problematic, "*I don't know what it* [CVS] *fixes, because vger has kept me out of the loop, and quite frankly I don't have the time to look at several hundred kilobytes of compressed patches by hand. And I refuse to apply patches that I don't feel comfortable with. ..And I'm going to ask David once again to just shut vger down, because these problems keep on happening*" (Torvalds, 1998 – Tues 29th

Sept)[1]. Torvalds' decision in 2002 to officially adopt a proprietary VCS for Linux development was met with dismay. Larry McVoy, a contentious figure, created and owned BitKeeper (BK) and had managed to convince Torvalds that BK would behave exactly the way Torvalds wanted for his Kernel. Table 1 summarizes key digital technologies used in open source development.

<< Insert Table 1 here >>

Open source developers rely upon digital technologies to collaborate. These tools archive content and conversations. VCS for instance ensures that the entire community becomes aware of which issues still need a software solution in the Kernel, and that no one's patch is overwritten by another developers' accidentally. VCS flags multiple fixes sent by different developers to resolve the same issue and demands a decision of the developer to choose which patch is superior. Most work is coordinated through a VCS but developers also have group as well as personal communications through IRC (Crowston and Howison 2005; German 2003; Jorgensen 2001). Email discussion forums are used regularly to air opinions, discuss development options, ask questions of their peers, and make announcements. The use of these digital technologies in combination makes open source development possible.

Data Collection

Our primary source of data was the Linux Kernel mailing list archive [LKML] that is kept up to date by the University of Indiana. We chose this site because it is a fairly exhaustive site dating as far back as June 1995. Its search engine allows full access to particular threads or phrases. At times other LKML sites were used to refine searches and points of reference when they offered some unique facilities which proved helpful when cross-checking that all the material had been collected and nothing on the topic of version control, and other technological tools used had been left out from the data set.

We accessed and collected data after a detailed reading of conversation threads in the LKML to ensure that we gathered the most important activity concerning version control software and technology use. We collected data chronologically to reduce the risk of missing highly relevant threads. Often one thread would

[1] http://www.uwsg.indiana.edu/hypermail/linux/kernel/9809.3/0766.html

break into another one that had a different title, and unless you followed the 'story' you could easily miss an important exchange. The exploratory search familiarized us to certain developers and keywords to further expand our data collection. The LKML site allowed the messages on each page to be sorted by date, subject and author. This feature was helpful, especially the *subject* sort because this broke all the messages down into their respective threads, thus making it possible to download each message related to every identified threads in the time period of June 1995 – June 2003. The URL's were saved for each email to ensure that we could return to the original text should the need arise. We constantly adapted our search keywords as the story unfolded. The main criterion for a message to be included into our data set was to have some mention of version control, communication technologies or related issues like VCS, BK, and IRC. This necessarily entailed reading a number of threads, which potentially could have been of use, but were later discarded for their lack of direct contribution to this study. Over time the data seemed to have been saturated "*whereby no additional data' could be found where the researcher could develop more properties*" (Glaser and Strauss 1967 p61).

We supplemented our data with peer-reviewed publications on the topic of version control software and digital technologies. An initial search through ISI Web of Science led us to 1,372 papers. Scanning these revealed that many papers listed were of limited relevance to our own work so we refined the search to include years 1996-2012 only. This reduced the number of hits to around 500 but, again, looking at the titles we noted how few were related to our work. The last stage of refining our search included using 'open source' as a search within our current search. We then returned 97 papers. We drew upon these papers to better grasp the technical work involved in open source software development work.

Our final data access point involved attending open source conferences where there was a revealing combination of software engineers, open source hackers, and computer science academics. We attended two such conferences. We observed how they interacted with each other while discussing different issues, some of which involved the technologies they used for every day development practice. If such a topic of discussion arose (e.g. during informal chats between developers or during formal presentations) we attempted to speak to the developers involved. We gathered data from developers in what we termed 'rapid interviews'. Such

interviews lasted between 5-20 minutes (average 12 minutes) and focused on a few questions. Most developers, when asked, agreed to short interviews and gave us quick and clear answers. We conducted 19 interviews in this manner. Table 2 summarizes our collected data.

<< Insert Table 2 here >>

Data Analysis

To manage our data we relied upon Qualitative Data Analysis software Atlas.ti. We coded email messages from the LKML and rapid interviews transcriptions (Phillips and Brown 1993). Our analysis methods involved a detailed study of reading and reflection (Butler 1998; Gadamer 1988; Prasad 2002) of multiple textual materials. Our approach made it possible to make sense of meaning within different texts but also across texts (Heracleous and Barrett 2001). The 97 academic publications we had studied served a mutually relevant but separate purpose of acquainting us much more deeply with relevant computer science background. Studying peer-reviewed papers related to version control software development and tools gave us the needed contextual knowledge to make better sense of conversations among developers, as well as of how various tools actually functioned. These publications aided us to 'go native' (Lok and de Rond 2013) in the open source developer world and to build the initial open code book (D'Adderio and Pollock 2014; Glaser and Strauss 1967; Van Maanen 1979).

All 3,352 messages were saved in a text file that was then used as a primary document for the content analysis software. We coded at various levels in order to fracture and expose the main themes in the data. Rather than code by word or phrase we looked to Strauss and Corbin (1999) who argued that coding entire sentences or paragraphs is a useful approach "*when the researcher already has several categories and wants to code specifically in relation to them*" (p. 120). This is not to say that more and new open codes were not generated during the open coding of the full email data collected but this coding scheme guided our analyses. The open coding procedure was accompanied by memo taking in the axial coding step. These memos developed into extended notes regarding emerging theoretical constructs as the analyses proceeded. Our data analysis drew close connections between the data and our theoretical objectives (Suddaby 2006). The first author had collected the data but the data analysis was followed in a constant comparative style through

numerous discussions between authors. Inspired by existing research where one of the researcher had been more 'native' than the others with regard to the data or different authors had focused on separate cases (Harrison and Rouse 2014), we established moments of discussion where the field researcher would defend her coding techniques and emergent ideas. After every few hundred messages that had been coded the authors would come together for a discussion (Glaser and Strauss 1967). They then questioned and cross-questioned the codes that had emerged, their significance and implications. A first process, of locking/cloning, emerged from such scrutiny and seemed particularly relevant because a) it occurred on a regular basis (multiple times in a day), b) was also a point of comparison between the two tools, CVS and BK, and c) emerged directly from the data.

Analyzing the locking/cloning process helped us conceptually distinguish various sub-elements, or early conceptual ideas. The field researcher then returned to the data armed with new codes that had been mutually developed by the authors. This set of codes looked for i) process triggers, ii) elements of folds, folding and unfolding, iii) examples of select developers gaining elite access, and iv) technological embeddedness of the fold. Drawing on these larger schemes for fracturing the data the field researcher was able to identify two additional complete processes of relevance. The data set had thus provided three key processes that were recurrent in the Linux Kernel case and seemed revealing of a process of folding and unfolding. The field researcher returned one more time to the data for another round of analysis, and was struck by a process that appeared 'broken' (her initial code). Further rounds of analysis and constant comparison subsequently brought in a fourth process, of 'ineffective folding and unfolding process' to the attention of the authors. Figure 1 visualizes the progression of our empirical analyses and the following section details the four resulting theoretically sampled processes.

We detail four folding and unfolding processes that emerged from our analyses in the following section.

FOLDING AND UNFOLDING PROCESSES IN LINUX DEVELOPMENT

This research examines how the transparency and openness of the development process may be navigated in the context of open source communities through overlaps in the use of and subtle differences in access to multiple digital technologies. A detailed analysis of our data gave us insight into how select developers used

several overlapping digital technologies in an overall open context to create segments of privileged, if momentary, secluded spaces. We will conceptualize infra such secluded spaces of work as digital folds. This section presents empirically how folds emerge through folding and are dissipated through unfolding. Developers temporarily inhabit a digital fold enabled by overlapping digital technologies, but maintaining the fold requires sustained effort on their part. The holding together of technologies by a developer to create a moment of isolation requires work and effort. When released, the folds dissipate. Unfolding brings the output of the fold (e.g. code, bug fixes, related documentation) back into the open. Unfolding often provides the only trace available of the existence of a digital fold. Unfolding generates an increase in the potential and scope of the abilities of technology being developed as well as in the open source community. Unfolding, through an amplification of resources and coordinating processes makes folds visible. Each process of digital folding below therefore comes with a corresponding unfolding.

This section details three processes whereby developers generated and dissipated digital folds. It also presents a fourth process that turned out to be incomplete and thus ineffective. For each of these processes we present the coalescing pressures that led to folding, the interplay among technologies, the digital space, its generativity, and the dissolution of the fold (see Table 3 for an analytical summary).

<< Insert Table 3 here >>

Process of Locking/Cloning - Generating privileged access to code

One of the most recurrent processes in our data was that of locking/cloning. The very ability to have your code accepted if you were a developer was dependent on access not only to the main code, but also to the mailing list, VCS and other digital technologies.

Coalescing pressures: Torvalds had managed a system through email and a few simple digital tools to isolate the kernel from the community. He would lock down his code to the rest of the developers by disallowing any 'logging facility'. Simple tools in this case, along with email use built a temporary moment of seclusion for Torvalds. This moment was necessary due to the large influx of patches sent to Torvalds and the growing pressure of being unable to cope unless some action was taken. Declaring his frustration about the state of Linux code on email he created a deliberate break in public discourse. His aim was to organize

his own ideas about Linux and fix current issues with the program with no space for interference from others. The process of locking the code down allowed him this moment where the various tensions in Linux could be addressed.

Interplay of multiple technologies: In CVS use this process was called locking but more contemporary version control software, like BitKeeper, calls this cloning. For the purpose of our paper we were interested in the link that was created by developers between Linux code and the CVS. We knew that in order to create this necessary interplaying link a developer would lock down the Linux code (the branch being worked on) and then begin to make changes to it. This forced a rupture in code development. Locking was done to stop other developers making a change at the same time whereby confusing the version control tool and breaking the Linux code. Locking was thus a key mechanism offered by version control software to sustain development with as few breakdowns in the Linux code as possible. CVS was used temporarily and by only a segment of the community. However, its use was controversial. Indeed Torvalds was clear in his disapproval of its use and explained it as a lack of control over his code. Many decisions were taken by him that he was reluctant to delegate to the community at large. This affected progress adversely but also the morale of the community,

> "*I don't like the idea of having developers do their own updates in my kernel source tree. I know that's how others do it, and maybe I'm paranoid, but there really aren't that many people that I trust enough to give write permissions to the kernel tree. Even people I have worked with for a long time I want to have the option of looking through their patches first, and maybe commenting on them (and I do reject patches from people)…Or I can decide (unilaterally) to revert a patch that results in problems*" (Torvalds, 1995)[2].

Digital space: We found that VCS and other tools were used to create privileged access to code for certain developers. In the case of BitKeeper this led to a very advanced method called push/pull and cloning. Linux code could be pulled by every developer as a cloned copy was created on their personal system. This would allow the developer to make changes while giving the same access and privilege to all other members of the community in real-time. However, levels of privilege differed when it came to the point of pushing changes back into the main code. BK grouped changes into changesets rather than force the developer to make single

[2] http://www.uwsg.indiana.edu/hypermail/Linux/kernel/9602/0800.html

changes, and these changesets enabled 'automatic synchronization' so that developers had an easy method of backtracking to specific changes rather than having to trace each single change and branching of the program. It also, somewhat like CVS, allowed for a copy of the master program to be made, called a clone in BK, and it had a push-pull method of control. However the key attraction of BK was its auto-merging algorithm, though there was still the possibility to perform manual merges as well. A developer could perform two-way as well as three-way merges with BK creating numerous overlaps in technologies.

> "*Linux BK doesn't depend on Linus using it. If it did, I wouldn't be using it. We've been tracking Linus for nearly a year, merging with him, taking patches from BK and non-BK users and it does work. You don't even need to track Linus yourself. You don't have to create a tree, either. Just clone the tree we've been maintaining to track Linus (2.2 or 2.4/2.3) and you have your own area to work in*" (Dougan, 2000)[3].

The work area suggested by Dougan is the orchestrated space built by digital tools where a developer is protected to work in solace and peace.

Generativity: Error messages, which are a common phenomenon with software development work, very importantly signify failure, but in our data analysis we found them to perform another role, that of building momentum and generative discussion, while marking the start of the unfolding of the digitally enabled secluded folds. They were necessary for the sustainability of code and this was evident from the generativity of coding activity and discourse that suddenly surrounded them,

> "Open source is good at debugging. AFTER the fact. People notice WHEN it breaks, not that it's GOING to break" (Landley, April 2002)[4].

An error message when running code drew the attention of the community to make repairs. Without error messages code development would make little progress. This simple yet essential mechanism of running the code to detect errors was thus one of the most generative and common practices in open source development. If developers simply provided read-only access then the possibility of generativity was reduced because though a problem could be noted it could not be changed or fixed by anyone other than the developer who had set the access rights.

[3] http://www.ussg.iu.edu/hypermail/linux/kernel/0009.1/1282.html
[4] http://lkml.iu.edu/hypermail/linux/kernel/0204.2/1334.html

Dissolution of the fold: The locking/cloning created folds where multiple technologies overlapped, but for these overlaps to be generative they necessitated an accompanying unfolding. In the case below the unfolding took the form of an (evident) breakdown:

> "*First, CVS doesn't fail silently. Second, I find it MUCH easier to figure out what CVS repository changes broke my local changes than to unpatch, get new source tree, repatch, and try to see what the error messages patch gave out as they fly off the screen. If the local patches were *not* to fail, a simple "cvs update" is much easier to perform than unpatch-getnewtree-repatch. When a local patch does fail, CVS leaves this really nice block in the file showing what repository and local changes overlapped. It's almost always very easy to fix the overlap. A failed merge will never compile because the compiler will give errors on the overlap markers. And a bad patch will never make it into the repository because only one person will have write access (and he'll only add Linus' approved patches)*" (Mason, 1996)[5].

Code being pushed back into the main repository was questioned by core developers that had maintainer roles in the community. This point would mark the dissolution of the temporary fold held together by the involved developer. The digital space evaporates and the repository is no longer invisibly locked to the rest of the community.

Process of Stealth Patching - Evading public review

A second process of folding/unfolding dealt with how digital technologies made it possible for some developers to evade public review of their work as well as slip their patch into the main code. Stealth patching, like locking and cloning is a phrase used in open source community development work (as well as in other software development environments). Here we shall not describe stealth patching in detail but rather delve into how some Linux developers used it to momentarily borrow multiple tools to slip patches into the main code without detection which obviates peer review.

Coalescing pressures: As mentioned stealth patching was made possible through a coalescing of pressures including a desire to contribute code, an available update created by a (usually) core developer, privileged access to different technical tools, and a desire to evade peer review (sometimes to simply expedite code development). Bits of code would thus be inserted into the final project without expected peer review because the developer had privileged access to other digital technologies. This involved a number of moments

[5] http://www.ussg.iu.edu/hypermail/linux/kernel/9602/0918.html

beginning with the key matter of aligning up multiple digital technologies for long enough to slip your code in but not too long that a break in code development is detectable.

Interplay of multiple technologies: In order to add a patch stealthily the developer involved would artfully, if only momentarily, align different technologies so that they would interface with each other. In Linux development, various developers used stealth patching over time. Especially revealing was how BitKeeper, as opposed to CVS, provided more opportunities for surreptitious technologies alignment. The data extract below shows that such alignment could even occur accidentally through BitKeeper thus making the Linux code vulnerable:

> "*With no public casualties (iput fuckup in 2.4.15 was an unrelated patch, but there was an idiotic bug that slipped into the patches sent to Linus and ate his tree - missed list_del() in a bad place ;-) And it involved complete rewrite of fs/super.c - including change of allocation rules, locking*" (Viro 2001)[6].

The alignment was between BK, the personal tree worked on by Torvalds, and a floating patch of software. This wreaked some havoc, which made it discernible. This was fortunate to us researchers because often stealth patches remain mostly invisible.

Digital space: The alignment of multiple technologies funneled an entrance into an un-gated space of submission within the version control software where furtive development could be carried out. The above example showed the extent of damage that could be provoked by a stealth patch yet often this process resonated better with purposeful, yet useful code submissions. The strength of the code patch could not be verified without peer review and testing, and this was why some Linux developers were irritated at code being slipped in:

> "*Who knows what code might slip through otherwise? They have to audit the entire revision history rather than just the patch they mean to send. That's a nightmare. The lawyers would never approve ANYTHING for release, except as a patch file*" (Landley 2002)[7].

A scrupulous process of code submission entailed less work fixing bugs at a later date. Open source development may well include the entire source code but the idea that it was easy to scrutinize it all after a certain size in lines of code was passed was not practical or easy.

[6] http://lkml.iu.edu//hypermail/linux/kernel/0112.3/0612.html
[7] http://lkml.iu.edu//hypermail/linux/kernel/0202.0/0257.html

Generativity: BitKeeper was criticized by some for making such patching more possible through its rather unique pull and push techniques of taking the Linux code and pulling it into a personal system, making changes to it, and then aligning your computer system with BK, to push your changes back into Linux code. Xymoron stressed above that BK seemed to have had an equal number of pushes to pulls. It was accepted knowledge that not all the developers in the Linux Kernel community had equal privilege with regard to code acceptance, and this was the basis of Xymoron's surprise at the identical number of pulls of code to pushes. Many, if not all had pull access but far fewer developers had push privileges. A code push would ideally undergo peer review. Peer review was considered by many to be essential to the quality of coding.

> *"Which patches are the stealth patches?*
> *The ones that say 'pull from here' are pretty opaque and seem to go past without much discussion. Off the top of my head, I'd say about I've seen about as many bk pushes as pulls but that could be perceptual bias*" (Garzik and Xymoron 2002)[8].

A discussion between two key developers in the Linux community above reflects the perception of stealth patching. This realization that a stealth patch had been attempted created a new thread of discussion amongst the Linux Kernel developers. Community rules of engagement and development practices were called into question by these and other developers. The invisibility of stealth patches usually would leave little trace but the community was beginning to find material traces of such practices and attempted to correct this issue. This discussion generated a change in practices that later translated into a redesign of the version control software to effect better detection.

Dissolution of the fold: The digital space dissolved when the alignment of technologies came to an end. The unfolding of the digital fold stressed the recognition that something had passed or changed. This was possible because a breakdown drew attention to the previously 'unknown' or when a change was noticed by a developer due to personal interest. The interchange below was intriguing because Daniel was annoyed by BK documentation being added to the Kernel tree. BK was a proprietary tool and it was decided by a majority vote in the Linux community to keep it as separate from Linux as possible. Some Linux developers feared that undue mixing between BK and Linux would not only tarnish the image of Linux as an open

[8] http://lkml.iu.edu//hypermail/linux/kernel/0204.2/1141.html

source project but actually defile its code. Daniel wanted to make the Linux community aware of BK documentation, which, according to him, had been stealthily added to Linux.

> "*Daniel is now bothered by the presence of BK documentation in the Linux kernel tree. Therefore, he submitted a patch to remove this documentation. Just about everybody else involved in this thread accuses him of censorship, for attempting to restrict the dissemination of ideas. I do not know whether all of these people use BK; all I know is the "censorship" claim, on the basis that he is restricting the dissemination of information*" (Stevie 2002)[9].

What became yet more interesting was that the unfolding of this stealth patch did indeed amplify the effect of BK documentation, but not quite as this developer might have imagined. The community, instead of perceiving him as a Linux and open source hero, accused him of censorship,

> "*Daniel was attempting a 'remove' operation... Daniel disagrees with the content of the speech in Documentation/BK-usage, based on his ideology. And he attempted to restrict the dissemination of that speech. What is the definition of censorship again? I see this as a clear cut case of Daniel's ideology pushing him to attempt to restrict speech...That is anti-freedom*" (Garzik 2002)[10].

Daniel's patch to remove BK documentation had been a stealth patch, not open to scrutiny, and thus problematic to the Linux community. It had been noticed and this was why there was even a discussion surrounding it causing its effect to intensify.

Process of Siphoning - Privileged policymaking

The recognition of the process of siphoning identifies the importance and relevance of real-time, privileged, and often face-to-face communication in open source development. Obviously open source software development would not be possible without the Internet and all the digital technologies it provides, yet open source developers are often geographically clustered in specific locations around the world (e.g. Silicon Valley). This geographical density does not translate into a non-reliance on digital technologies, but it is interesting all the same. The ability to privilege some developers over others is apparent in many open source projects. We found this very true in our Linux Kernel case.

Coalescing pressures: Broad changes to development practices became necessary from time to time. Some were triggered by the processes detailed above while others involved mistakes and breakdowns. Any such

[9] http://lkml.iu.edu//hypermail/linux/kernel/0204.2/1105.html
[10] http://lkml.iu.edu//hypermail/linux/kernel/0204.2/1004.html

instance would build anxiety and pressure within the community prompting the more experienced and elite members of a community (the core) to recognize the need to discuss new policies, code direction changes and other serious decisions pertinent to the community and code.

Interplay of multiple technologies: The process of siphoning involved a break in public discourse. This was made possible through an alignment of more than one digital technology to create some level of private space. We found that many Linux developers made abundant use of IRC with email and version control software that made more exclusive, invitation-only communications possible even in an open environment:

> "*Ben's right. Most patches are independent because the work divides itself up that way, because people talk about this stuff (on IRC) and cooperate, and because the tree structure evolves to support the natural divisions ;)*
> *In a fan club, saying "andrea's the MM guy, talk to him" is only natural. It's a meritocracy, he's alpha geek on call in that area right now*" (Landley and Phillips 2001)[11].

This email conversation illustrated how accepted IRC use was within the community, even though its limitations to public access and visibility were recognized. Certain developers argued against the use of such folding technologies yet most acknowledged that they played a key role in advancing the development process. Siphoned spaces created a tighter and small space where a few developers could pause to discuss controversial matters and come to a decision. It was accepted that open communities built collaborative environments for debate but there were also occasions that required a quick decision, where the different emails from the larger community created 'noise' rather than helpful discussion,

> "*If Linus and the top dozen lieutenants all had special scripts and encryption keys set up (all using open source software) so that their code got to each other's systems more easily and was looked at first before shoveling through the signal to noise ratio on lkml, or the random spam Linus gets daily*" (Landley, April 2002)[12].

This and other moments indicated that open source communities needed public spaces for open discussion but at the same time reserved digital folds a role to play in sustaining a community.

Digital space: Siphoned spaces created privileged and elite-only access to high level decision making in the Linux community. Developers not privy to accessing these siphoned decision-making spaces questioned the application level of decisions taken. This invalidated the basic principles and 'law' of open source for them

[11] http://lkml.iu.edu//hypermail/linux/kernel/0201.0/0004.html
[12] http://lkml.iu.edu/hypermail/linux/kernel/0204.2/1334.html

as they insisted that openness brought more scrutiny, and thus better code, and development. This developer argued below to make the development process more open, but more importantly, all major decisions related to development to be held in public online settings as well:

> "*I'd like to see the development process carried out more in the open and to that extent, increasing reliance on Bitkeeper, with its convenient point-to-point push/pull paths is worrisome. What we have now is, *everybody* with a piece of kernel to maintain is in on the private, point-to-point thing. It's efficient, no doubt, but I fear we're also weakening one of one the basic driving forces of Linux development, that is, the public debate part... you won't find a lot of design discussion* [on the lkml postings]… *even though… many changes are taking place that will affect kernel development for years to come. It used to be that every major change would start with an [RFC]. Now the typical way is to build private consensus between a few well-placed individuals and go straight from there to feeding patches.*
> *Not voting, discussion. Without the discussion we miss the chance to get thousands of eyeballs on the issue*" (Phillips 2002)[13].

Generativity: The issues discussed in siphoned spaces could be seen to steer impact and often substantial change in the community. The example above showed awareness amongst developers of confidential discussions being held. Our analysis revealed that the implications of such discussions for the development process may not be simply categorized as useful or harmful. They were often a mix of both as they had different implications for various members of the developer community,

> "*I have to observe that the current process is *not broken* in the sense that development is now proceeding at a truly impressive rate… thus removing the immediate pressure to perform….True, I haven't noticed a lot of grumbling about dropped patches lately*" (Phillips, April 2002)[14].

Phillips explained that whatever had been approved amongst the core developers in the siphoned space seemed to be working well as code development was beginning to show substantial progress. The generative nature of folding/unfolding was evident because traceable changes in code were visible.

Dissolution of the fold: Private discussions would have remained concealed in Linux if no traces had been left behind. The fact that developers summoned or encouraged other developers to a less visible IRC or email conversation indicated to others that a break did occur even if there was little disclosure of what had been discussed. Code, bits of development and even developers were thus temporarily quarantined into

[13] http://lkml.iu.edu/hypermail/linux/kernel/0204.2/1224.html
[14] http://lkml.iu.edu/hypermail/linux/kernel/0204.2/1224.html

privileged spaces. These moments of silence created an 'absence' that left little trace yet was made 'visible' because it created a break in the usual fabric of community conversation. The texture of the communal discussion changed from one of question, crisis, rebuke, and thwarting control or disagreement to one of resolution, partial or otherwise. But inevitably such siphoned discussions needed to be rewoven back into the community discussion so that the new decisions could take effect,

> "*Yes, it came out in the course of the thread - Linus and Jeff had a private email exchange in which Linus had Jeff to push his Bitkeeper documentation files into the tree*" (Phillips, April 2002)[15].

The conversation between Linus and Jeff was not visible to the community but a trace of it remained because it was mentioned by one of them in the LKML discussion forum. The fold created by Linus to speak in isolation with Jeff was brief and dissolved the moment the email was successfully in Jeff's inbox. The simple technologies used to create the digital fold involved an email programme, email server, and POP3/IMAP but as soon as the email was directed to the correct recipient the fold was dissolved.

Partial/Ineffective Folding

What happened when the process of folding/unfolding was for some reason thwarted and failed to produce a generative digital space was especially intriguing. Such situations provided us with the opportunity to contrast effective folding/unfolding with that of an ineffective or incomplete process. Our data revealed that the folding process of locking/cloning when disrupted drew the lack of opacity in development work to the notice of the developer community. Particularly revealing, developers did not consider reduced opacity as troubling per se. Rather, they were concerned about the breakdown in their ability to work. From the various discussions amongst the developers during such breakdowns it became evident that developers recognized the role and privilege of core developers and it is the pragmatics of coding that they concern themselves with rather than hierarchy questions,

> "*Linus has separated the maintainers list into two layers because he cream-skimmed out a half dozen lieutenants in charge of major subsystems. Those lieutenants have a direct hotline to Linus, and the maintainers are expected to filter their patches through them. Individual contributors filter their patches through the maintainers, then the lieutenants, then Linus.*

[15] http://lkml.iu.edu/hypermail/linux/kernel/0204.2/1224.html

> *This is not a bad thing, it means that by the time Linus sees code it's been code reviewed by two people: one with intimate knowledge of the particular subsystem and the other with broader knowledge of other areas it needs to interoperate with. And this hierarchy, now that people know about it, is probably equally as responsible for the declogging of the patch queue as Bitkeeper is*" (Landley, April 2002)[16].

Coalescing pressures: The desire to fix a bug or make some changes to Linux code coupled with the ability to fix the issue and a possible solution came together to create pressure. This convinced the developer to upload the desired solution to the bug, and have his name added to the list of Linux Kernel contributors. Knowledge that he had a possible solution to the problem encouraged him to make the attempt to upload his work. This involved turning his attention to gaining access to more than one technology to make the change happen.

Interplay of multiple technologies: Folding and unfolding occur numerous times in community development work, but not all folding processes happen fully. An example in Linux development where a fold was attempted but that didn't materialize can be seen in the discussion between Manolov and McVoy. Manolov, a Linux developer, was attempting to make a pull of code from the version control software, BitKeeper, but found that he had been locked out. His inability to create a link with BK made it impossible for him to finish his work. The fold was not possible though he needed it to do his work, and manage the various links to multiple technologies.

> *"It looks like we have a bad disk, I'm checking them now to figure out if it is the the primary or backup data drive. I'll run checks in all the repositories if fsck doesn't find the problem so it may take a couple of hours before we are back up"* (McVoy, March 2002)[17].
> *"I can't pull from linux-2.[45] and i'm getting:*
> *ERROR-Lock fail: possible permission problem.*
> *Last time I got this error somebody was playing with the config files"* (Manolov, March 2002)[18].

Digital space: In this particular example the fold was made impossible because of a failure in BitKeeper. As McVoy explained in this email thread he was unsure if the breakdown was in the primary or the back-up drive. Not all the content was backed up so he wanted to fix and then mirror the repositories to ensure completeness. Version control software, in this case BK, is often a crucial part of the fold when code changes

[16] http://lkml.iu.edu/hypermail/linux/kernel/0204.2/1334.html
[17] http://lkml.iu.edu/hypermail/linux/kernel/0203.3/0590.html
[18] http://lkml.iu.edu/hypermail/linux/kernel/0203.3/0691.html

and updates are concerned. Its breakdown made folding impossible. Manolov was therefore unable to create a digital space to work in and he could not make any updates to Linux either. If any partial digital space had been created it was not enough, or was not appropriate for Manolov to be able to complete his development task.

Generativity: The breakdown in the folding process became visible to the community and caused an outcry. The community did not complain about the opacity of work in open source development but, rather, about the breakdowns in the folding process,

> "*I wouldn't be a bit surprised if we have some permissions problems… we'll fix things as we become aware of them. In fact, I know we have permissions problems… There are a couple of trees which are missing files, both in Rik's linuxvm.bkbits.net, I suspect an interrupted clone. They are:*
>
> *bk://linuxvm.bkits.net/linux-2.5-vmtidbits*
>
> *bk://linuxvm.bkits.net/linux-2.5-writethrot*
>
> *Rik, ping me if you need help cleaning these up.*
>
> *The ppc tree seems to be missing linuxppc_2_4, Paul/Tom/Troy/Cort, where is this tree? You'll want to get a copy back.* " (McVoy, March 2002)[19].

This episode was illustrative of how open source developers tacitly accepted the dual need for transparency and opacity. The generative elements of this folding/unfolding can be illustrated by McVoy's alacrity to resolve Manolov's lock-out. McVoy's email replies were shot out speedily to show good faith to the community and his desire to make BitKeeper functional again.

> "*Leaving the drive off overnight "fixed it" enough that I am able to get some of the data off. It will be a couple hours before I know how much, but I did manage to get all the ssh keys, project descriptions, and project statistics. I'm now working on the actual data just in case there is one of the trees, such as the ppc trees, that we can't find again. The drive has bad blocks and when it hits them it goes into retry la la land, so I won't know which data is bad until I hit the bad blocks"* (McVoy, March 2002)[20].

Dissolution of the fold: The attempt made by Manolov to pull code from BK was not productive so the digital space that he wanted to create never emerged. If any partial fold was created, this was dissolved by Manolov so that he could return to the use of the public email group to establish a reason for the breakdown.

[19] http://lkml.iu.edu/hypermail/linux/kernel/0203.3/0689.html
[20] http://lkml.iu.edu/hypermail/linux/kernel/0203.3/0779.html

In so doing he created a trace to his ineffective folding which made the entire community aware of the problem and, it could be argued, led to less aggravated developers faced with the same problem.

These processes of folding and unfolding in Linux development work, including the ineffective one, helped illuminate key elements of how open source communities develop software and coordinate work. Some ideas that struck us in the findings were how developers were constantly in a process of balancing their need to finish working on code (pragmatics) with the often stated principles of keeping discussions open and transparent to all (ideology of open source development). We return to this and other implications after having elaborated upon the nature of digital folds and the process of folding and unfolding.

THEORETICAL DEVELOPMENT

This section draws from the empirical analyses and the conceptual foundations of this work to build a theoretical understanding: of digital folds as protected digital spaces for work; of the process of folding and unfolding that enables the creation and dissolution of digital folds; and of how this process balances dynamically openness and transparency with closure and opacity in open source communities (see Figure 2).

<< Insert Figure 2 here >>

Digital Folds

We define a digital fold as the drawing together of multiple digital technologies created and sustained by actors in times of coalescing pressures to create a momentary digital space of work. It is a folding from what happens outside, in the larger world and community, into a more secluded inside that generates a temporary pocket of stability and retreat for reflective organizing (Deleuze 1992; Deleuze and Strauss 1991; Kavanagh and Araujo 1995). We are inspired by Deleuze's ideas and by Kavanagh and Araujo's accessible interpretation of how technologies can foster capsules that offer those 'inside' some momentary measure of a more quiet space. As such folds are temporary, they elapse and bring those inside back into the larger environment of the community.

Folds are temporary for they require some work to be maintained. Folds are thus not permanent, and nor can they occur or endure without human effort. Select developers who draw upon several overlapping technologies create an interfacing position that constitutes the virtual locus for the fold. Most of these

technologies are used as tools, in isolation as well as in conjunction, to do development work and it requires effort on the part of the developers to make them interface for any length of time.

Digital folds are elusive and hidden to many in the community. These spaces are places of work, conversations, and reflection that are not publicly visible. Digital spaces make isolation and interaction possible but at the same time are not inclusive. Visibility of the space and access for a developer has to be earned through reputation based upon contributions to the communally developed product (Dahlander and Frederiksen 2012; Howison and Crowston 2014; Lave and Wenger 1991; Setia et al. 2012; von Krogh et al. 2003). The developers that are privy to such spaces of isolated work are then the core, established few. It is a position of privilege to be able to create a fold where more private code development and discussion can be done. The peripheral developers or actors on a project spend months and in some cases years to attain the necessary knowledge and expertise to be able to contribute, and thus become a core developer. Digital spaces of work are about creating an escape from the cacophony of too many diverse ideas, opinions, and range of experiences that makes communication stilted (Kelty 2008).

Developers who have less privileged access to tools and code are less able to create or hold a fold with certain technologies like version control software. The nature of the fold is thus affected by the acknowledged expertise levels of a developer but also by the nature of the tools and technologies being harnessed to create the fold. Some technologies are designed to exclude and include access to actors differently. Version control software was usually built and designed to have access control but far more importantly to have editing control.

Folding and Unfolding in Open Source Development

Digital folds have a fluid nature where change(ing) is the norm. They are enabled by the use of digital technologies and represent therefore a virtual rather than a tangible space. Folds' fluidity stems from the work carried out by the developers to create and maintain the fold. Work is needed to build an interface between different technologies in order to access information and content necessary for coding. It is in the very playing and use of multiple technologies that virtual spaces are created. This is the opacity and closure that is made possible. Each of these technologies may be accessible freely to all members of the community.

However, some actors move from one technology to the next as their coding work demands. This generates opacity in the process through closure to certain tools and an inability to update code. Most of the community developers will not follow this handful of isolated developers from platform to platform, and retracing their steps is difficult. This is due in large part to faint digital traces that reduce visibility. Faint digital traces may occur because some tools and technologies used like libraries are accessed by too many developers, multiple times and often simultaneously leading to obscurity over which developer is in the process of accessing which tool(s), and when. Distinguishing one user from another becomes difficult when attempting to catch and become a part of a digital fold. Other reasons for faint digital traces include the specific nature of the technologies involved in the fold. Barriers to access are easily created in some tools like version control software. Multiple technologies use is predicated on the reliance of such developers on more than one technology to complete even the simplest of algorithms that can then be seen as a contribution (i.e. is an accepted piece of code that fixes previous problems and does not break the system build).

Unfolding is the natural continuation of the process. This is not another or separate process, it is the release of the fold to effectuate a difference. The work or discussion that is established within the fold is given a release through subsequent unfolding. Unfolding amplifies the effect of the fold through its release and the accompanying change, be that a change in code or decision after a discussion. It makes the change visible, but the change itself was made possible by the entire folding and unfolding process.

Actors work to create and maintain the fold but such folds are not easy to sustain because the defining qualities of open source software development are openness and transparency. They need constant work to hold together. The release of a fold or its dissolution returns the community development to greater transparency and openness, and the private digital space evaporates. The unfolding therefore entails the dissolution of the digital fold. Unfolding involves the actors releasing the interfacing links they have created between different and multiple technologies. The process of dissolving can be captured analytically in a number of ways, for example when a new patch of code appears and is included in the most current codebase.

The process of folding and unfolding is generative because it: creates a new type of momentum in the development process; generates a temporary protected digital space of work with overlapping technologies;

releases any change in code or decision back into the public domain of community development; and leads to new potentialities and ideas when the changes are brought to the attention of other developers and there is initiation of a new discussion.

Balancing openness and transparency with closure and opacity, dynamically

Openness and transparency are qualities that are both enabling and constraining for open source development. In a similar fashion opacity and closure are both enabling and constraining; what they enable and constrain is symmetrically opposed. Transparency in open source development work makes the performance of development inclusive and close to the constitutional ideas of open source ensconced in the licence (Cornford et al. 2010; Weber 2004). Likewise, openness enables all members of the community to be able to read the code, follow the process of development, recommend changes and bug fixes and carry on discussions of code merit. At the same time such discussions can lead to an impasse and make decision-making close to impossible. There are multiple view points on the strengths of various algorithms, version direction (to list a few) and this is understandable considering development work is done jointly and each member will naturally have his/her own biases and preferences. These pressures arise due to such impasses. The process of folding and unfolding navigates the pressures brought by these qualities. When transparency and openness lead to a dead-end in the development process some developers may create a protected space of refuge to work through an idea to overcome the impasse. In the same manner at some point when opacity and closure become too complete, and protected developers may well have exhausted their ability to work further on their ideas and will then seek feedback from the larger community. We argue that the process of folding and unfolding maintains the enabling dimensions of openness and transparency while also managing the constraints they generate for open source community work. Pragmatically, thus, openness and transparency, just as closure and opacity, are not given once and for all but are navigated by developers on an ongoing basis.

IMPLICATIONS

The theorizing of folding and unfolding in open source development was inspired by our empirical analyses but holds implications beyond the single case of Linux Kernel development to open source and online work

scholarship. Our research brings attention to a notion of relevance and importance to many open and transparent contexts, that of managing the balance between the rush of ideas and solutions offered by the entire community and the need for smaller, protected and quiet spaces of work. Our theory of folding and unfolding provides insight into how such a balance is achieved to help select participants negotiate development work within a larger community. Specifically, this research holds implications for open source communities research: by conceptualizing digital folds; by articulating how folding and unfolding help combine the ideology of openness and transparency with the pragmatic need for moments of greater opacity and closure to get the job done; and, by conceptualizing further important and so far little examined dimensions of open source work.

The Nature of Digital Folds

This research adds to scholarship a conceptualization of digital folds in open source work. It therefore holds implications for research that has examined related phenomena of secluded spaces of work in various environments, in particular, Vaan et al. (2015) and Vedres and Stark's (2010) "structural folds" and Kellogg's (2009) "relational spaces." Vedres and Stark (2010) considered what happens in overlapping networks, i.e. in structural folds of one network onto another. They argued that structural folds have the potential to be innovative because tightly knit groups develop a familiarity to access the resources necessary to build knowledge and work together. Vaan et al (2015) further considered that in structural folds clashes and discord caused by diversity in expertise and (mis-)communication generate new ideas and innovation. Structural folds rely on bringing experts from diverse networks together into a fold so that they are forced to communicate (Vaan et al. 2015; Vedres and Stark 2010). On a related but distinct note, building upon ethnographic work rather than social network analysis, and considering the condition for institutional change in organizations, Kellogg's (2009) conceptualized relational spaces as areas of isolation and inclusion where actors with opposing interests came together to stimulate organizational change.

Our research relates to these influential concepts but also differs from them in important manners and in doing so extends theory on secluded spaces of work. Structural folds (Vaan et al. 2015; Vedres and Stark 2010) rely on bringing experts from diverse groups together into a fold so that they are forced to

communicate. According to Vaan et al. (2015) and Vedres and Stark (2010), experts who happen to be situated in structural folds are forced to communicate with each other, which can lead to innovation. The digital folds conceptualized in this research do not contradict the notion of structural fold but instead address the importance, in open source development at least, of private spaces that do not welcome diversity of expertise but, rather, encourage the deep focus of select experts on a specific issue.

Moreover, akin to relational spaces (Kellogg 2009), the digital folds conceptualized in this research make developers' isolation and interactions possible. However, dissimilar to relational spaces, digital folds are not inclusive. In digital folds, access has to be earned through reputation-based contributions to the communally developed product (Bergquist and Ljungberg 2001; Grewal et al. 2006; Masum 2001; Stewart 2005). The developers that are privy to such protected spaces of work are then the core and established few. It is a position of privilege to be able to generate a digital fold where more private code development and discussions happen. Peripheral developers or actors on a project may spend months and in some cases years to attain the necessary knowledge and expertise to be able to contribute, and thus become a core developer. Kellogg's relational spaces highlighted the importance of inclusiveness and openness to diverse groups of people. By contrast, digital folds are far more about an escape from the cacophony of too many diverse ideas, opinions, and range of experience that can at times slow down the collective development process in open source.

Pragmatically respecting the principles of openness and transparency

The discourse of open source software communities emphasizes the defining principles of openness and transparency (Hertel et al. 2003; Lakhani and Wolf 2005). Some have even argued that ideology, along with rituals, culture and tradition, is a key part of what brings the developers together to form a community (Choi et al. 2015). At the same time, open source developers' pragmatism has also been recognized (David and Shapiro 2008; Stewart and Gosain 2006; Torvalds and Diamond 2001). Faced with a choice between following principles or practicality, they may choose the latter (Torvalds and Diamond 2001). Apparent dilemmas between espoused principles of openness and transparency and pragmatism manifest when the need to complete work and build an effective codebase clashes with principles. In such situations, open source

developers at times even use proprietary software and tools for key areas of development such as version control software (Shaikh and Cornford 2003; Weber 2004). Open source, therefore, relies and builds on the premise of openness and transparency. However, we note how such ideological beliefs deepen and shift when a community begins to take more formal shape over time. Ideology then also encompasses meritocracy, earned privilege for developers, levels of access rights and a reputation earned through solid code contributions. Research in the area of open source ideology implies a certain necessary undulation in getting community agreement on any issue (Stewart and Gosain 2006) too quickly. An over emphasis on implementation of decisions only coming *after* an agreement has been reached may indeed create a solidifying process that can retard task completion (Stewart and Gosain 2006, p307). In a large, disparate community an agreement needs to arise out of the noise and chaos of multiple viewpoints (Bergquist and Ljungberg 2001; Casadesus-Masanell and Ghemawat 2006; Raymond 1999), and this could continue indefinitely if there are no mechanisms put into place to create a resolution. The process of folding and unfolding can be understood as a way to bring about resolutions while maintaining fluidity. This process showcases that successful open source communities like the Linux Kernel one are more fluid in how agreement is reached and in when decisions are implemented. At the same time ideological ideas of privilege and access control play an influential role in bringing agreement.

Ideology, as can be noted from different elements encompassed under this term (Stewart and Gosain 2006) is also not a 'binary phenomenon' (Choi et al. 2015, p683). Diverse open source communities understand this idea differently as do individual developers within communities. Certain communities have the reputation of being strongly ideological while others are seen as pragmatic. The Linux Kernel case was fascinating in this regard as its leader was strongly pragmatic but a large part of those in the community referred equally strongly to the ideology of open source.

Open source communities however do not solely rely upon the principles of openness and transparency, but also upon that of meritocracy. A reliance on the principles of meritocracy is a mechanism by which ideological differences are actually negotiated in open source communities. As we had noted supra, the meritocratic principle relates to how status is achieved in open source community work. Meritocracy is the

underlying mechanism by which developers gain the reputation of a good programmer and achieve status (O'Mahony and Ferraro 2007; Roberts et al. 2006). If a developer works hard and well, and her contributions are peer-reviewed to be significant then such a developer is able to yield status in the community (Bergquist and Ljungberg 2001). The meritocratic mechanism relies on experience, expertise and hard work as reputation signifiers but at the same time this makes privileged access to community tools and code possible. Privileged access is then less questioned by the rest of the community because they respect the process by which the developer has earned it (Stewart 2005).

The folding and unfolding process and the notion of digital fold conceptualized in this work illuminate how, pragmatically, the meritocratic principle helps balance out the principles of openness and transparency in open source communities. This research thus adds to current scholarship on open source communities by showing how the espoused values of openness, transparency, and meritocracy can still dominate the development process while opacity and privileged access actually operate, at least intermittently, and further cement the contributions of core developers over time. Digital folds, and the restricted access to such spaces, are made possible from prior meritocratic rise of select developers who have already proven their value in the open and are then able to sustain their status in the community through privileged access to overlapping digital technologies. These privileged developers can work in protected and provisional digital folds, which provide a pragmatic solution to problems arising in the open, broader, community and respect the espoused principles of openness and transparency.

How open source communities work

There have been some insights in the literature concerning the changes wrought in open source development, both in the nature of the community (Shaikh 2015) and in the process of development due to greater commercialization (Fitzgerald 2006). Research has in particular already showed nuanced changes over time in open source ideology and community values (Feller and Fitzgerald 2002; Fitzgerald 2005; Fitzgerald 2006; Fitzgerald and Feller 2002). This paper adds to such scholarship on open source communities work by being less focused on examining the changes in open source communities over time than in understanding more deeply a so far less examined aspect of development work in open source.

Moreover, there has been substantial scholarship on the private-collective model as a way to understand how open source borrows from both the private and collective forms of innovation at the individual and collective level, which helps make sense of developer motivations and of how work is organized (von Hippel and von Krogh 2003; Von Hippel and Von Krogh 2006; von Hippel and von Krogh 2016; von Krogh et al. 2012; von Krogh et al. 2003). This stream of research has examined how the more elite developers in open source projects are able to reap higher private benefits. Such ideas are clearly echoed in this work as well. Our study complements and extends this scholarship on the private-collective model by providing a micro analysis of how elite developers work, retain their positions of influence, and obtain non-pecuniary benefits by examining how such developers manipulate and control multiple technologies.

This research further adds to scholarship on work in open source communities by theorizing how certain developers develop the ability to work temporarily in quiet enclaves of work by using multiple technologies. This research unpacked the on-going navigation between transparency and opacity and between openness and closure in relation to the collective process of writing code. It revealed how some developers can rely upon overlapping technologies in ways that reflect their achieved privileged status within the community to create opacity which then in turn allows them space to write and upload code in peace. The process of folding and unfolding highlights the dynamic relationships in open source development between developers' specialization, privilege, and reputation status and their subtly differentiated access to software building technologies.

Furthermore, this research also adds to scholarship on open source communities' work by highlighting an unanticipated consequence of the reliance upon multiple technologies. That open source developers rely upon multiple technologies is not in itself surprising or new. At some points in the development process developers rely upon multiple tools since no singular customized and comprehensive tool may fulfil all developers' needs all the time. This study however highlighted a more unexpected consequence of the reliance upon multiple digital tools: certain developers can manipulate the availability of various tools simultaneously in ways that reflect their earned privilege and provides them with some temporary protection, even as these tools are equally available to any and all developers of the community. The de

facto creation of pockets of opacity and closure that stems from the reliance upon - and mastery of - multiple digital tools is particularly intriguing because it pragmatically benefits select developers' work while also staying true to the rhetoric and ideology of openness and transparency.

CONCLUSION

Complete, uninterrupted transparency and openness are not always the most generative qualities when multiple opinionated experts are all eager to be heard and followed. They at times need to be balanced out with more closure and opacity in open source communities. Our theoretical contribution lies in reflecting on how moments of reduced transparency and openness, digital folds, emerge within an open source community's development work through folding and dissolve through unfolding.

There are limitations to this work that warrant further investigations. For one, this research uncovered how digital folds helped some developers solve tricky issues and accomplish temporarily isolated work. It is however possible that digital folds be used for other purposes for open source communities as well. Future research could thus examine if digital folds may be used for reasons other than to complete delicate work in open source communities. Also, this study highlighted how the use of multiple technologies enabled the emergence of digital folds. However, digital folds are also likely to appear in other conditions. What such other conditions are and how they may be activated in digital folds would be interesting for future research to investigate. Moreover, the nature and roles of multiple digital technologies in use simultaneously in open source communities merits research in its own right. Our study touches on this aspect but only in relation to digital folds. Future research that focuses on the complexity of multiple technologies use with regard to security concerns, software breakdowns, and even unplanned actions on the part of technology become imperative in a world with growing dependence on software. Finally, an especially fascinating avenue for scholarship would be to investigate the interrelationship among multiple digital folds and their possible implications for development practices.

REFERENCES

Bagozzi, R.P., and Dholakia, U.M. 2006. "Open Source Software User Communities: A Study of Participation in Linux User Groups " *Management Science* (52:7), pp. 1099-1115.

Bar, M., and Fogel, K. 2003. *Open Source Development with Cvs*. Scottsdale, AZ: Paraglyph Publishing.

Barrett, M., Heracleous, L., and Walsham, G. 2013. "A Rhetorical Approach to It Diffusion: Reconceptualizing the Ideology-Framing Relationship in Computerization Movements," *MIS Quarterly* (37:1), pp. 201-220.

Ben-Menahem, S., von Krogh, G., Erden, Z., and Schneider, A. 2015. "Coordinating Knowledge Creation in Multidisciplinary Teams: Evidence from Early-Stage Drug Discovery," *Academy of Management Journal*), June 16, 2015.

Benkler, Y. 2002. "Coase's Penguin, or, Linux and the Nature of the Firm," *Yale Law Journal* (112:3), pp. 369-446.

Benkler, Y. 2004. "Sharing Nicely: On Shareable Goods and the Emergence of Sharing as a Modality of Economic Production," *Yale Law Journal* (114:273-358).

Bergquist, M., and Ljungberg, J. 2001. "The Power of Gifts: Organising Social Relationships in Open Source Communities," *Information Systems Journal* (11:4), pp. 305-320.

Berliner, B. 1990. "Cvs Ii: Parallelizing Software Development," *Proceedings of the USENIX Winter 1990 Technical Conference*, Washington D.C.

Bonaccorsi, A., and Rossi, C. 2003. "Why Open Source Software Can Succeed," *Research Policy* (32:7), pp. 1243-1258.

Boudreau, K. 2010. "Open Platform Strategies and Innovation: Granting Access Vs. Devolving Control," *Management Science* (56:10), pp. 1849-1872.

Butler, T. 1998. "Towards a Hermeneutic Method for Interpretive Research in Information Systems," *Journal of Information Technology* (13), pp. 285-300.

Campbell-Kelly, M., and Garcia-Swartz, D.D. 2009. "Pragmatism, Not Ideology: Historical Perspectives on Ibm's Adoption of Open-Source Software," *Information Economics and Policy* (21:3), pp. 229-244.

Casadesus-Masanell, R., and Ghemawat, P. 2006. "Dynamic Mixed Duopoly: A Model Motivated by Linux Vs. Windows," *Management Science* (52:7), pp. 1072-1084.

Choi, N., Chengalur-Smith, I., and Nevo, S. 2015. "Loyalty, Ideology, and Identification: An Empirical Study of the Attitudes and Behaviors of Passive Users of Open Source Software," *Journal of the Association for Information Systems* (16:8).

Clegg, S., Kornberger, M., and Rhodes, C. 2005. "Learning/Becoming/Organizing," *Organization* (12:2), pp. 147-167.

Clemm, G. 1989. "Replacing Version-Control with Job-Control," *Proceedings of the 2nd International Workshop on Software configuration management*, Princeton, New Jersey, United States, pp. 162-169.

Coleman, G. 2004. "The Political Agnosticism of Free and Open Source Software and the Inadvertent Politics of Contrast," *Anthropological Quarterly* (77:3), pp. 507-519.

Corbet, J., Kroah-Hartman, G., and McPherson, A. 2013. "Linux Kernel Development: How Fast It Is Going, Who Is Doing It, What They Are Doing, and Who Is Sponsoring It ", Linux Foundation.

Cornford, T., Shaikh, M., and Ciborra, C. 2010. "Hierarchy, Laboratory and Collective: Unveiling Linux as Innovation, Machination and Constitution," *Journal of the Association for Information Systems* (11:11).

Crowston, K., and Howison, J. 2005. "The Social Structure of Free and Open Source Software Development.," *First Monday*).

Crowston, K., and Howison, J. 2006. "Hierarchy and Centralization in Free and Open Source Software Team Communications," *Knowledge, Technology, and Policy* (18:4), Winter, pp. 65-85.

D'Adderio, L., and Pollock, N. 2014. "Performing Modularity: Competing Rules, Performative Struggles and the Effect of Organizational Theories on the Organization," *Organization Studies*), August 19, 2014.

Dabbish, L., Stuart, C., Tsay, J., and Herbsleb, J. 2012. "Social Coding in Github: Transparency and Collaboration in an Open Software Repository," in: *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*. Seattle, Washington, USA: ACM, pp. 1277-1286.

Dafermos, G. 2001. "Management and Virtual Decentralized Networks: The Linux Project," *First Monday* (11:6).

Daffara, C., and Gonzalez-Barahona, J.M. 2010. "Open Source Software for Open Government Agencies," in *Open Government,* D. Lathrop and L. Ruma (eds.). Sebastopol, CA: O'Reilly Media, pp. 345-362.

Dahlander, L. 2007. "Penguin in a Newsuit: A Tale of How De Novo Entrants Emerged to Harness Free and Open Source Software Communities," *Industrial and Corporate Change* (16:5), pp. 913-943.

Dahlander, L., and Frederiksen, L. 2012. "The Core and Cosmopolitans: A Relational View of Innovation in User Communities," *Organization Science* (23:4), pp. 988-1007.

Dahlander, L., and Gann, D.M. 2010. "How Open Is Innovation?," *Research Policy* (39:6), pp. 699-709.

Dahlander, L., and O'Mahony, S. 2011. "Progressing to the Center: Coordinating Project Work," *Organization Science* (22:4), pp. 961-979.

David, P.A., and Shapiro, J.S. 2008. "Community-Based Production of Open-Source Software: What Do We Know About the Developers Who Participate?," *Information Economics and Policy* (20:4), pp. 364-398.

Deleuze, G. 1992. *The Fold: Leibniz and the Baroque*. University of Minnesota Press.

Deleuze, G., and Strauss, J. 1991. "The Fold," *Yale French Studies* (80), pp. 227-247.

Deodhar, S.J., Saxena, K.B.C., Gupta, R.K., and Ruohonen, M. 2012. "Strategies for Software-Based Hybrid Business Models," *The Journal of Strategic Information Systems* (21:4), pp. 274-294.

Economides, N., and Katsamakas, E. 2006. "Two-Sided Competition of Proprietary Vs. Open Source Technology Platforms and the Implications for the Software Industry," *Management Science* (52:7), pp. 1057-1071.
Eisenhardt, K.M., and Graebner, M.E. 2007. "Theory Building from Cases: Opportunities and Challenges," *Academy of Management Journal* (50:1), pp. 25-32.
Endres, M.L., Endres, S.P., Chowdhury, S.K., and Alam, I. 2007. "Tacit Knowledge Sharing, Self-Efficacy Theory, and Application to the Open Source Community," *Journal of Knowledge Management* (11:3), 2007/06/05, pp. 92-103.
Felin, T., and Zenger, T.R. 2014. "Closed or Open Innovation? Problem Solving and the Governance Choice," *Research Policy* (43:5), pp. 914-925.
Feller, J., and Fitzgerald, B. 2000. "A Framework Analysis of the Open Source Software Development Paradigm," *The 21st International Conference in Information Systems (ICIS 2000)*, Brisbane, pp. 58-69.
Feller, J., and Fitzgerald, B. 2002. *Understanding Open Source Software Development*. London, UK: Addison-Wesley.
Feller, J., Fitzgerald, B., and van der Hoek, A. 2002. "Open Source Software Engineering," *IEE Proceedings - Software* (149:1), pp. 1-2.
Fitzgerald, B. 2005. "Has Open Source a Future?," in *Perspectives on Free and Open Source Software,* J. Feller, B. Fitzgerald, S. Hissam and K. Lakhani (eds.). Cambridge, MA: MIT Press, pp. 121-140.
Fitzgerald, B. 2006. "The Transformation of Open Source Software," *MIS Quarterly* (30:3), September, 2006, pp. 587-598.
Fitzgerald, B., and Feller, J. 2002. "A Further Investigation of Open Source Software: Community, Co-Ordination, Code Quality and Security Issues," *Information Systems Journal* (12:1), pp. 3-6.
Flyvbjerg, B. 2006. "Five Misunderstandings About Case-Study Research," *Qualitative Inquiry* (12:2), April 1, 2006, pp. 219-245.
Fogel, K. 1999. *Open Source Development with Cvs*. Scottsdale, AZ: Coriolis Open Press.
Gacek, C., and Arief, B. 2004. "The Many Meanings of Open Source," *Software, IEEE* (21:1), pp. 34-40.
Gadamer, H.G. 1988. "On the Circle of Understanding," in *Hermeneutics Versus Science? Three German Views,* J.M. Conolly and T. Keutner (eds.). IN: University of Notre Dame Press.
German, D.M. 2003. "The Gnome Project: A Case Study of Open Source, Global Software Development," *Software Process: Improvement and Practice* (8:4), pp. 201-215.
Glaser, B.G., and Strauss, A. 1967. *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Chicago: Aldine.
Grewal, R., Lilien, G.L., and Mallapragada, G. 2006. "Location, Location, Location: How Network Embeddedness Affects Project Success in Open Source Systems," *Management Science* (52:7), 2006/07/01, pp. 1043-1056.
Grune, D. 2003. "Concurrent Versions System Cvs." 2004, from http://www.cs.vu.nl/~dick/CVS.html#History
Gurstein, M.B. 2011. "Open Data: Empowering the Empowered or Effective Data Use for Everyone?," *First Monday* (16:2).
Harrison, S., and Rouse, E. 2014. "Let's Dance! Elastic Coordination in Creative Group Work: A Qualitative Study of Modern Dancers," *Academy of Management Journal* (57:5), December 6, 2013, pp. 1256-1283.
Hars, A., and Ou, S. 2002. "Working for Free? Motivations for Participating in Open-Source Projects," *International Journal of Electronic Commerce* (6:3), pp. 25–39.
Heracleous, L., and Barrett, M. 2001. "Organizational Change as Discourse: Communicative Actions and Deep Structures in the Context of Information Technology Implementation," *Academy of Management Journal* (44:4), pp. 755-778.
Hertel, G., Niedner, S., and Herrmann, S. 2003. "Motivation of Software Developers in Open Source Projects: An Internet-Based Survey of Contributors to the Linux Kernel," *Research Policy: Special Issue on Open Source Software Development* (32:7), pp. 1159-1178.
Hoegl, M., Weinkauf, K., and Gemuenden, H.G. 2004. "Interteam Coordination, Project Commitment, and Teamwork in Multiteam R&D Projects: A Longitudinal Study," *Organization Science* (15:1), pp. 38-55.
Howison, J., and Crowston, K. 2014. "Collaboration through Open Superposition: A Theory of the Open Source Way," *MIS Quarterly* (38:1), pp. 29-50.
Janssen, M., Charalabidis, Y., and Zuiderwijk, A. 2012. "Benefits, Adoption Barriers and Myths of Open Data and Open Government," *Information Systems Management* (29:4), 2012/09/01, pp. 258-268.
Jorgensen, N. 2001. "Putting It All in the Trunk: Incremental Software Development in the Freebsd Open Source Project," *Information Systems Journal* (11), pp. 321-336.
Kavanagh, D., and Araujo, L. 1995. "Chronigami: Folding and Unfolding Time.," *Accounting, Management and Information Technology* (5:2), pp. 103-121.
Kellogg, Katherine C. 2009. "Operating Room: Relational Spaces and Microinstitutional Change in Surgery," *American Journal of Sociology* (115:3), pp. 657-711.
Kelty, C.M. 2008. *Two Bits: The Cultural Significance of Free Software*. Duke University Press.
Kilpi, T. 1997. "New Challenges for Version Control and Configuration Management: A Framework and Evaluation," *IEEE Computer*:1st Euromicro Working Conference on Software Maintenance and Reengineering (CSMR '97)), March, pp. 33-41.

Koch, S., and Schneider, G. 2000. "Results from Software Engineering Research into Open Source Development Projects Using Public Data," *Tätigkeitsfeld Informationsverarbeitung und Informationswirtschaft,* H.R. Hansen and W.H. Janko (eds.).
Koch, S., and Schneider, G. 2002. "Effort, Cooperation and Coordination in an Open Source Software Project: Gnome," *Information Systems Journal* (12:1), pp. 27-42.
Kogut, B., and Metiu, A. 2001. "Open-Source Software Development and Distributed Innovation," *Oxford Review of Economic Policy* (17:2), pp. 248-264.
Krishnamurthy, S., and Tripathi, A.K. 2009. "Monetary Donations to an Open Source Software Platform," *Research Policy* (38:2), pp. 404-414.
Lakhani, K., and von Hippel, E. 2003. "How Open Source Software Works: "Free" User-to-User Assistance," *Research Policy* (32), pp. 923-943.
Lakhani, K.R., and Wolf, R.G. 2005. "Why Hackers Do What They Do: Understanding Motivation and Effort in Free/Open Source Software Projects," in *Perspectives on Free and Open Source Software,* J. Feller, B. Fitzgerald, S. Hissam and K.R. Lakhani (eds.). MIT Press.
Laurent, A.M.S. 2004. *Understanding Open Source and Free Software Licensing*. Sebastopol, CA: O'Reilly
Lave, J., and Wenger, E. 1991. *Situated Learning : Legitimate Peripheral Participation*. Cambridge England ; New York: Cambridge University Press.
Lee, G.K., and Cole, R.E. 2003. "From a Firm-Based to a Community-Based Model of Knowledge Creation: The Case of the Linux Kernel Development," *Organization Science* (14:6), pp. 633-649.
Lerner, J., and Tirole, J. 2002. "Some Simple Economics of the Open Source," *The Journal of Industrial Economics* (2), pp. 197-234.
Lerner, J., and Tirole, J. 2005. "The Scope of Open Source Licensing," *Journal of Law, Economics, and Organization* (21:1), pp. 20-56.
Ljungberg, J. 2000. "Open Source Movements as a Model for Organizing," *European Journal of Information Systems* (9:4), pp. 208-216.
Lok, J., and de Rond, M. 2013. "On the Plasticity of Institutions: Containing and Restoring Practice Breakdowns at the Cambridge University Boat Club," *Academy of Management Journal* (56:1), February 1, 2013, pp. 185-207.
MacCormack, A., Rusnak, J., and Baldwin, C.Y. 2006. "Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code," *Management Science* (52:7), pp. 1015-1030.
Masum, H. 2001. "Reputation Layers for Open Source Development.," *Making Sense of the Bazaar: Proceedings of the 1st Workshop on Open Source Software Engineering.,* J. Feller, B. Fitzgerald and A. van der Hoek (eds.).
Moody, G. 2001. *Rebel Code: Linux and the Open Source Revolution*. London: Penguin.
O'Mahony, S., and Ferraro, F. 2007. "The Emergence of Governance in an Open Source Community," *Academy of Management Journal* (50), pp. 1079-1106.
O'Reilly, T. 1999. *Open Sources: Voices from the Open Source Revolution*. Boston: O'Reilly.
O'Reilly, T. 2010. "Government as a Platform," in *Open Government,* D. Lathrop and L. Ruma (eds.). O'Reilly Media, pp. 11-39.
Oh, W., and Jeon, S. 2007. "Membership Herding and Network Stability in the Open Source Community: The Ising Perspective," *Management Science* (53:7), pp. 1086-1101.
Olson, M. 2005. "Dual Licensing," in *Open Sources 2.0: The Continuing Evolution,* C. DiBona, M. Stone and D. Cooper (eds.). Sebastopol, CA: O'Reilly, pp. 71-90.
Oram, A. 2011. "Promoting Open Source Software in Government: The Challenges of Motivation and Follow-Through," *Journal of Information Technology & Politics* (8:3: Special Issue: The Politics of Open Source), pp. 240-252.
Osterloh, M., and Rota, S. 2007. "Open Source Software Development—Just Another Case of Collective Invention?," *Research Policy* (36:2), pp. 157-171.
Phillips, N., and Brown, J.L. 1993. "Analyzing Communication in and around Organizations: A Critical Hermeneutic Approach," *Academy of Management Journal* (36:6), December 1, 1993, pp. 1547-1576.
Powell, A. 2012. "Democratizing Production through Open Source Knowledge: From Open Software to Open Hardware," *Media, Culture & Society* (34:6), September 1, 2012, pp. 691-708.
Prasad, A. 2002. "The Contest over Meaning: Hermeneutics as an Interpretive Methodology for Understanding Texts," *Organizational Research Methods* (5:1), January 1, 2002, pp. 12-33.
Raymond, E. 1999. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. Sebastopol, California: O'Reilly & Associates.
Raymond, E.S., and Trader, W.C. 1999. "Linux and Open-Source Success," *IEEE Software* (16:1), pp. 85-89.
Roberts, J., Hann, I.-H., and Slaughter, S. 2006. "Understanding the Motivations, Participation, and Performance of Open Source Software Developers: A Longitudinal Study of the Apache Projects," *Management Science* (52:7), pp. 984-999.
Scacchi, W., and Alspaugh, T.A. 2012. "Understanding the Role of Licenses and Evolution in Open Architecture Software Ecosystems " *Journal of Systems and Software* (85:7), pp. 1479–1494.
Schweik, C. 2003. "The Institutional Design of Open Source Programming: Implications for Addressing Complex Public Policy and Management Problems," *2003* (8:8), p. January.
Setia, P., Rajagopalan, B., Sambamurthy, V., and Calantone, R. 2012. "How Peripheral Developers Contribute to Open-Source Software Development," *Information Systems Research* (23:1), pp. 144–163.

Shah, S.K. 2006. "Motivation, Governance, and the Viability of Hybrid Forms in Open Source Software Development," *Management Science* (52:7), pp. 1000-1014.
Shaikh, M. 2015. "Embedding Penguins into the Company: Sourcing the 'Right' Sauce " *Academy of Management Annual Meeting*, Vancouver, Canada.
Shaikh, M., and Cornford, T. 2003. "Version Management Tools: Cvs to Bk in the Linux Kernel," *25th International Conference on Software Engineering - Taking Stock of the Bazaar: The 3rd Workshop on Open Source Software Engineering,* J. Feller, B. Fitzgerald, S.A. Hissam and K. Lakhani (eds.), Portland, Oregon, pp. 127-132.
Sharma, S., Sugumaran, V., and Rajagopalan, B. 2002. "A Framework for Creating Hybrid-Open Source Software Communities," *Information Systems Journal* (12:1), pp. 7-26.
Singh, P.V., and Phelps, C. 2013. "Networks, Social Influence, and the Choice among Competing Innovations: Insights from Open Source Software Licenses," *Information Systems Research* (24:3), pp. 539-560.
Spaeth, S., von Krogh, G., and He, F. 2014. "Research Note—Perceived Firm Attributes and Intrinsic Motivation in Sponsored Open Source Software Projects," *Information Systems Research* (0:0), p. null.
Spaeth, S., von Krogh, G., and He, F. 2015. "Research Note—Perceived Firm Attributes and Intrinsic Motivation in Sponsored Open Source Software Projects," *Information Systems Research* (26:1), pp. 224-237.
Stallman, R. 1984. "The Gnu Manifesto." from http://www.gnu.org/gnu/manifesto.html
Stallman, R. 1999a. "The Free Software Definition." 2003, from http://www.gnu.org/philosophy/free-sw.html
Stallman, R. 1999b. "The Gnu Operating System and the Free Software Movement," in *Open Sources : Voices from the Open Source Revolution,* C. DiBona, S. Ockman and M. Stone (eds.). O'Reilly.
Stallman, R.M. 2002. *Free Software, Free Society: Selected Essays of Richard M. Stallman*. Lulu.com.
Stewart, D. 2005. "Social Status in an Open-Source Community," *American Sociological Review* (70:5), October 1, 2005, pp. 823-842.
Stewart, K.J., Ammeter, A.P., and Maruping, L.M. 2006. "Impacts of License Choice and Organizational Sponsorship on User Interest and Development Activity in Open Source Software Projects," *Information Systems Research* (17:2), pp. 126-144.
Stewart, K.J., and Gosain, S. 2006. "The Impact of Ideology on Effectiveness in Open Source Software Development Teams," *MIS Quarterly* (30:2), June, pp. 291–314.
Streeter, L.A., Kraut, R.E., Lucas, H.C., and Caby, L. 1996. "How Open Data Networks Influence Business Performance and Market Structure," *Communications of the ACM* (39:7 (July)), pp. 62-73.
Suddaby, R. 2006. "What Grounded Theory Is Not," *Academy of Management Journal* (49), pp. 633-642.
Torvalds, L. 1999. "The Linux Edge," in *Open Sources: Voices from the Open Source Revolution,* C. DiBona, S. Ockman and M. Stone (eds.). Sebastopol, California: pp. 101-111.
Torvalds, L., and Diamond, D. 2001. *Just for Fun: The Story of an Accidental Revolutionary*. Harper Collins.
Tullio, D.D., and Staples, D.S. 2014. "The Governance and Control of Open Source Software Projects " *Journal of Management Information Systems* (30:3), pp. 49-80
Vaan, M.d., Vedres, B., and Stark, D. 2015. "Game Changer: The Topology of Creativity," *American Journal of Sociology* (120:4), pp. 1144-1194.
Van Maanen, J. 1979. "The Fact of Fiction in Organizational Ethnography," *Administrative Science Quarterly* (24), pp. 539-550.
van Oorschot, K.E., Akkermans, H., Sengupta, K., and Van Wassenhove, L.N. 2013. "Anatomy of a Decision Trap in Complex New Product Development Projects," *Academy of Management Journal* (56:1), February 1, 2013, pp. 285-307.
Vedres, B., and Stark, D. 2010. "Structural Folds: Generative Disruption in Overlapping Groups," *American Journal of Sociology* (115:4), pp. 1150-1190.
von Hippel, E., and von Krogh, G. 2003. "Open Source Software and the "Private-Collective" Innovation Model: Issues for Organization Science," *Organization Science* (14:2), March-April, pp. 209-223.
Von Hippel, E., and Von Krogh, G. 2006. "Free Revealing and the Private-Collective Model for Innovation Incentives," *R&D Management* (36:3), pp. 295-306.
von Hippel, E., and von Krogh, G. 2016. "Crossroads—Identifying Viable "Need–Solution Pairs": Problem Solving without Problem Formulation," *Organization Science* (27:1), pp. 207-221.
von Krogh, G., Haefliger, S., Spaeth, S., and Wallin, W. 2012. "Carrots and Rainbows: Motivation and Social Practice in Open Source Software Development," *MIS Quarterly* (36:2), pp. 649-676.
von Krogh, G., and Spaeth, S. 2007. "The Open Source Software Phenomenon: Characteristics That Promote Research," *Journal of Strategic Information Systems* (16:3), pp. 236-253.
von Krogh, G., Spaeth, S., and Lakhani, K.R. 2003. "Community, Joining, and Specialization in Open Source Software Innovation: A Case Study," *Research Policy* (32:7), July, pp. 1217-1241.
von Krogh, G., and von Hippel, E. 2006. "The Promise of Research on Open Source Software," *Management Science* (52:7), pp. 975-983.
Weber, S. 2004. *The Success of Open Source*. Harvard University Press.
West, J. 2003. "How Open Is Open Enough? Melding Proprietary and Open Source Platform Strategies," *Research Policy* (32), pp. 1259-1285.

West, J. 2007. "Seeking Open Infrastructure: Contrasting Open Standards, Open Source and Open Innovation," *First Monday, Peer Reviewed Journal on the Internet* (12:6).
West, J., and Bogers, M. 2014. "Leveraging External Sources of Innovation: A Review of Research on Open Innovation," *Journal of Product Innovation Management* (31:4), pp. 814-831.
West, J., and Gallagher, S. 2006. "Challenges of Open Innovation: The Paradox of Firm Investment in Open Source Software," *R&D Management* (36:3), pp. 315-328.
Wright, A.L., and Zammuto, R.F. 2013. "Wielding the Willow: Processes of Institutional Change in English County Cricket," *Academy of Management Journal* (56:1), February 1, 2013, pp. 308-330.
Yin, R. 1981. "The Case Study Crisis: Some Answers. ," *Administrative Science Quarterly* (26), pp. 58-65.

TABLES AND FIGURES

Table 1: Digital Technologies in Open Source

Name of Technology	Acronym	Definition
Version control software	VCS	Is a digital tool by which multiple versions of any software can be managed, kept track of and protected against overwriting.
Concurrent Versions System	CVS	Is a type of version control software which is open source in nature. It was created by Dick Grune in 1986 but has since evolved by Brian Berliner and others.
BitKeeper	BK	Another version control system but this one is proprietary. It is a distributed VCS created by Larry McVoy and his company, BitMover in 2000.
Internet Relay Chat	IRC	Is an electronic application that facilitates textual based communication in both group and private one-on-one settings.
Libraries		A set of resources in the form of code, sub-routines and help data that are pre-written and can be drawn upon by computer programmes to help build software (Wikipedia).
Linux Kernel Mailing List	LKML	Is an email group subscribed to by all the Linux Kernel developers where the majority of discussions about development are held, and archived.
Linux Kernel	LK	Central and most essential part of an operating system that coordinates activities on a computer system. Linux Kernel is the Unix-based central software that runs within and to coordinate the GNU Hurd.

Table 2: Data Sources

Data Access Point	Data Features	Use in Analysis
Primary data source		
Linux Kernel Mailing List (LKML)	- 3352 emails - Single-spaced printed pages = 1892	Deepened our knowledge and insight into digital folding and overlaps and focused our attention on the implications stemming from such tools creating new binds to facilitate communication. Gave us direct access and insight into the facets of digital technologies used by online communities, and the tensions they create and dissipate with a historical, longitudinal level of access.
Secondary data sources		
Academic publications (Web of Science)	- 1372 articles on version control software - 97 articles were focused on open source	Developed and enhanced our understanding of the different kinds of digital technologies in use by open source communities. It sensitized us to nuances in language and functionality in preparation for coding the LKML data. It facilitated our ability to parse technical, hacker conversations.
Conference observations and interviews	- 19 rapid interviews - OSS Conference 2007-2012 - Open World Forum 2008-2011	Triangulate facts and observations: Developed and enhanced our understanding of how developers work, collaborate and coordinate daily development work. Gave us direct access and insight into the facets of digital technologies used by online communities, and the tensions they create and dissipate.

Table 3: Folding/Unfolding in Linux Development				
Concepts and processes	**Locking/Cloning**	**Stealth patching**	**Siphoning**	**Partial/Ineffective folding**
Coalescing pressures	Tension created through a disparity between the need for control evinced by the elite developers of the community, and the community's desire to keep the code base growing	Privileged access to technical tools coupled with a desire to reduce peer review of personal bug fix/code	Experienced and elite members of a community (the core) recognize the need to discuss new policies, code direction changes and other serious decisions pertinent to the community and code	Tension created through a disparity between the need for control evinced by the elite developers of the community, and the community's desire to keep the code base growing
Interplay of multiple technologies	Bridging a link between code, and email through a lock in the version control software creating a rupture in code development	Artful and momentary alignment of digital tools to push changes	Decisive demand or need to break the discourse in public through access to invitation-only tools	Attempting to bridge a link between code, and email through a lock in the version control software creating a rupture in code development
Digital space	Creation of privileged and exclusive space for development	Entrance to un-gated submission space within version control software that allows for furtive development	Creation of privileged and elite-only access to high level decision making space	*The digital space required is not created*
Generativity	Failure in code integration and breakdown in software propagating ideas, solutions, discussion and improvement	Realization of stealth patches led to a re-evaluation of community rules of engagement and development	Discussions held in siphoned spaces steered impact and change in the community	Failure in code pull and breakdown in access to code creates unease in the community which generates attempts to rectify the problem
Dissolution of the fold	Decisive push of code back to the communal code questioned through digital gatekeepers	Reweaving the code and discourse to reconnect to the community	Reweaving the final decision and/or code through limited discourse to reconnect to the community	Dissolution of any partial fold leaving behind a trace of the failed attempt
Folding	The process of aligning multiple technologies by a (core) developer to create an exclusive digital space for work that allows for generative change that becomes visible when the fold dissipates.			The *ineffective* process of aligning multiple technologies so that a digital space is not possible and generative change cannot be engineered.
Unfolding	The natural follow through of folding where the digital space created through the folding process is dissolved to reveal decisions, change, and the trace of folding itself.			A digital space was not created as required thus rendering unfolding as unnecessary and less generative.

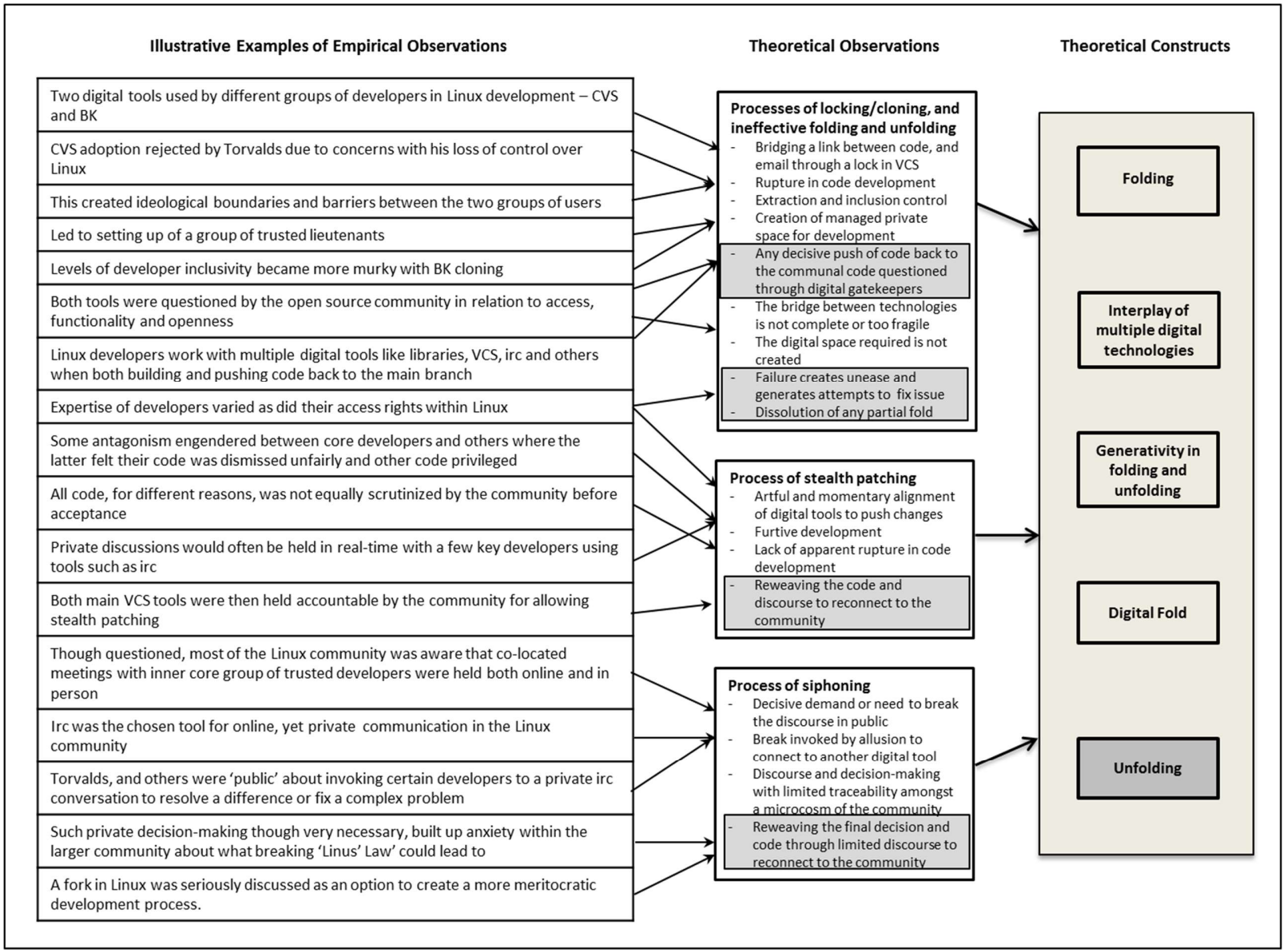

Figure 1: Data Analysis and Theoretical Constructs

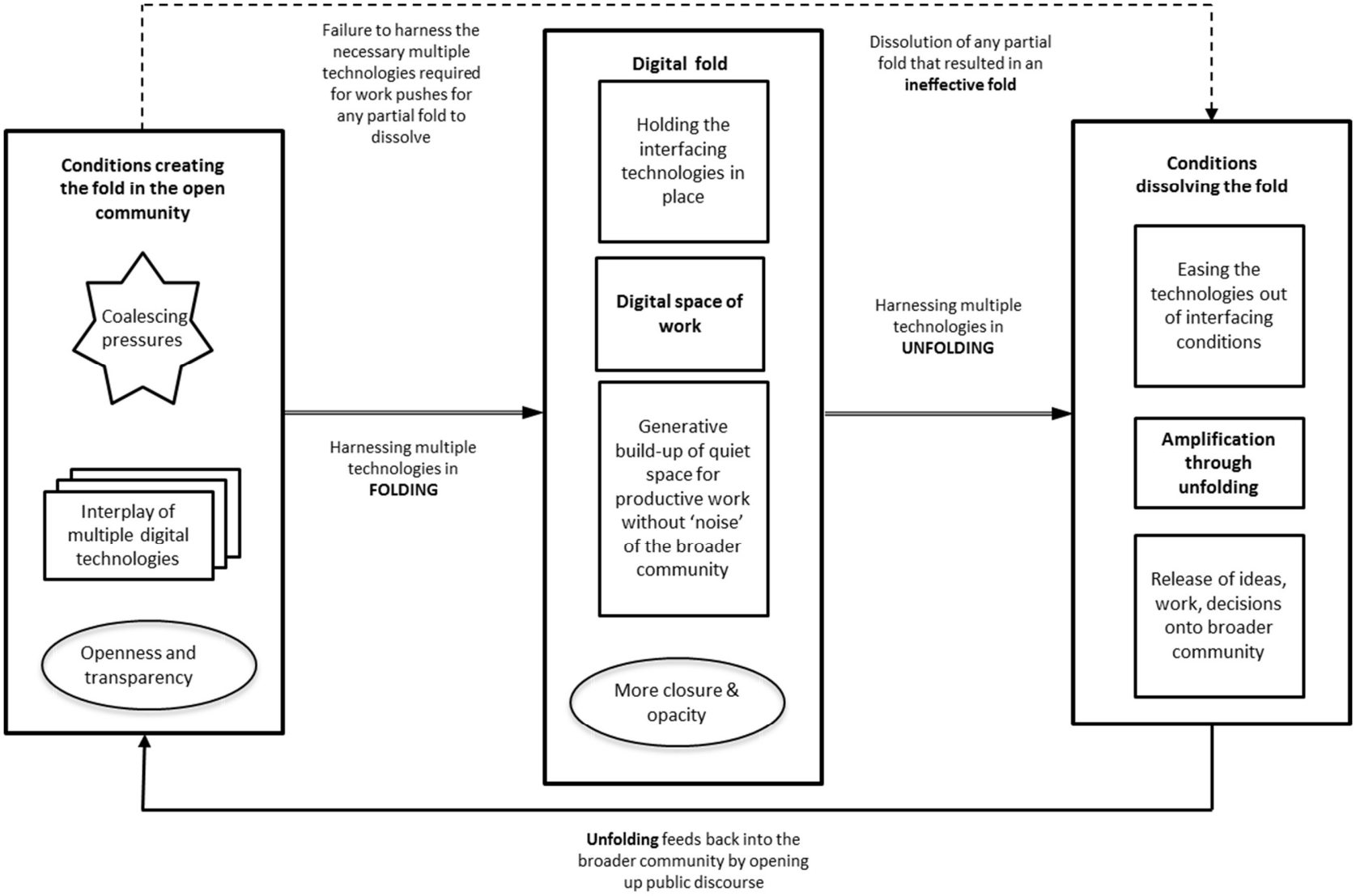

Figure 2: Theoretical Model of Folding and Unfolding